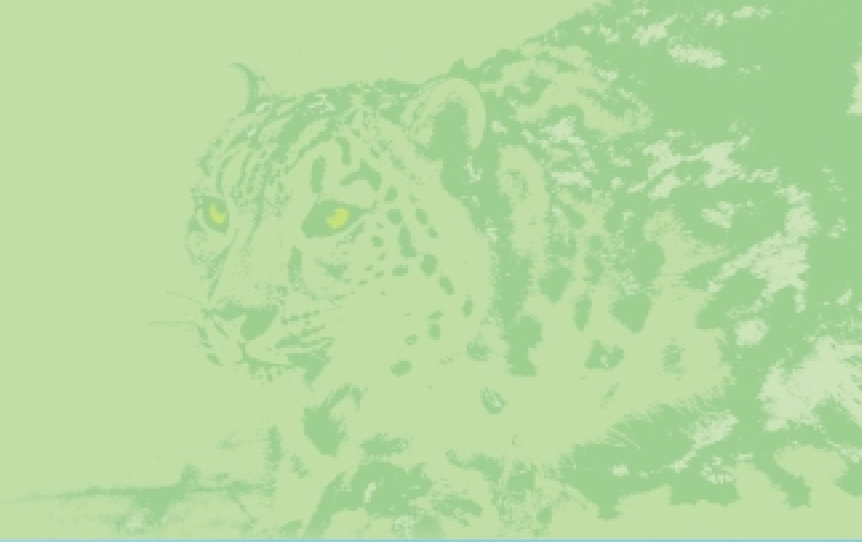


CHAPTER 9

Exception Handling

9 Exception Handling

It's the exception that proves the rule.

Common saying

Introduction

One way to divide the task of designing and coding a method is to code two main cases separately: the case where nothing unusual happens and the case where exceptional things happen. Once you have the program working for the case where things always go smoothly, you can then code the second case where exceptional things can happen. In Java, there is a way to mirror this approach in your code. You write your code more or less as if nothing very unusual happens. After that, you use the Java exception handling facilities to add code for those exceptional cases.

The most important use of exceptions is to deal with methods that have some special case that is handled differently depending on how the method is used. For example, if there is a division by zero in the method, then it may turn out that for some invocations of the method the program should end, but for other invocations of the method something else should happen. Such a method can be defined to throw an exception if the special case occurs, and that exception will allow the special case to be handled outside of the method. This allows the special case to be handled differently for different invocations of the method.

throw exception

handle exception

In Java, exception handling proceeds as follows: Either some library software or your code provides a mechanism that signals when something unusual happens. This is called **throwing an exception**. At another place in your program you place the code that deals with the exceptional case. This is called **handling the exception**. This method of programming makes for cleaner code. Of course, we still need to explain the details of how you do this in Java.

Prerequisites

Almost all of this chapter only uses material from Chapters 1 through 5 and Chapter 7. The only exception is the subsection "`ArrayIndexOutOfBoundsException`," which also uses material from Chapter 6. However, that subsection may be omitted if you have not yet covered Chapter 6. Chapter 8 is not needed for this chapter.

9.1 Exception Handling Basics

Well the program works for most cases. I didn't know it had to work for that *case.*

Computer Science Student, *Appealing a grade*

Exception handling is meant to be used sparingly and in situations that are more involved than what is reasonable to include in an introductory example. So, we will teach you the exception handling details of Java by means of simple examples that would not normally use exception handling. This makes a lot of sense for learning about the exception handling details of Java, but do not forget that these first examples are toy examples and, in practice, you would not use exception handling for anything that simple.

Example

A Toy Example of Exception Handling

Display 9.1 contains a simple program that might, by some stretch of the imagination, be used at a dance studio. This program does not use exception handling, and you would not normally use exception handling for anything this simple. The setting for use of the program is a dance lesson. The program simply checks to see if there are more men than women or more women than men and then announces how many partners each man or woman will have. The exceptional case is when there are no men or no women or both. In that exceptional case, the dance lesson is canceled.

try block

In Display 9.2 we have redone the program using exception handling. The keyword try labels a block known as the try **block**, which is indicated in the display. Inside the try block goes the non-exceptional cases and checks for the exceptional cases. The exceptional cases are not handled in the try block, but if detected they are signaled by "throwing an exception." The following three lines taken from inside the multiway if-else statement are the code for throwing the exception:

```
throw new Exception("Lesson is canceled. No students.");
throw new Exception("Lesson is canceled. No men.");
throw new Exception("Lesson is canceled. No women.");
```

If the program does not encounter an exceptional case, then none of these statements that "throw an exception" is executed. So, in that case we need not even know what happens when an exception is "thrown." If no exception is "thrown," then the code in the section labeled "catch block" is skipped and the program proceeds to the last statement, which happens to output "Begin the lesson." Now, let's see what happens in an exceptional case.

throwing an exception

If the number of men or the number of women is zero (or both), that is an exceptional case in this program and results in an exception being **thrown**. To make things concrete, let's say that the

Display 9.1 **Handling a Special Case without Exception Handling *(Part 1 of 2)***

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DanceLesson
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader keyboard =
                  new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Enter number of male dancers:");
        String menString = keyboard.readLine();
        int men = Integer.parseInt(menString);

        System.out.println("Enter number of female dancers:");
        String womenString = keyboard.readLine();
        int women = Integer.parseInt(womenString);

        if (men == 0 && women == 0)
        {
            System.out.println("Lesson is canceled. No students.");
            System.exit(0);
        }
        else if (men == 0)
        {
            System.out.println("Lesson is canceled. No men.");
            System.exit(0);
        }
        else if (women == 0)
        {
            System.out.println("Lesson is canceled. No women.");
            System.exit(0);
        }

       // women >= 0 && men >= 0
       if (women >= men)
            System.out.println("Each man must dance with " +
                                     women/(double)men + " women.");
        else
            System.out.println("Each woman must dance with " +
                                      men/(double)women + " men.");
        System.out.println("Begin the lesson.");
    }
}
```

Later in this chapter we will finally explain this.

Display 9.1 **Handling a Special Case without Exception Handling** ***(Part 2 of 2)***

Sample Dialogue 1

```
Enter number of male dancers:
4
Enter number of female dancers:
6
Each man must dance with 1.5 women.
Begin the lesson.
```

Sample Dialogue 2

```
Enter number of male dancers:
0
Enter number of female dancers:
0
Lesson is canceled. No students.
```

Sample Dialogue 3

```
Enter number of male dancers:
0
Enter number of female dancers:
5
Lesson is canceled. No men.
```

Sample Dialogue 4

```
Enter number of male dancers:
4
Enter number of female dancers:
0
Lesson is canceled. No women.
```

number of men is zero, but the number of women is not zero. In that case the following statement is executed, which is how Java throws an exception:

```
throw new Exception("Lesson is canceled. No men.");
```

Let's analyze this statement. The following is the invocation of a constructor for the class `Exception`, which is the standard Java package `java.lang`.

```
new Exception("Lesson is canceled. No men.");
```

Display 9.2 **Same Thing Using Exception Handling *(Part 1 of 2)***

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DanceLesson2
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader keyboard =
                   new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Enter number of male dancers:");
        String menString = keyboard.readLine();
        int men = Integer.parseInt(menString);

        System.out.println("Enter number of female dancers:");
        String womenString = keyboard.readLine();
        int women = Integer.parseInt(womenString);

        try
        {
            if (men == 0 && women == 0)
                throw new Exception("Lesson is canceled. No students.");
            else if (men == 0)
                throw new Exception("Lesson is canceled. No men.");
            else if (women == 0)
                throw new Exception("Lesson is canceled. No women.");

            // women >= 0 && men >= 0
            if (women >= men)
                System.out.println("Each man must dance with " +
                                      women/(double)men + " women.");
            else
                System.out.println("Each woman must dance with " +
                                      men/(double)women + " men.");
        }
        catch(Exception e)
        {
            String message = e.getMessage();
            System.out.println(message);
            System.exit(0);
        }

        System.out.println("Begin the lesson.");
    }

}
```

This is just a toy example to learn Java syntax. Do not take it as an example of good typical use of exception handling.

try block

catch block

Display 9.2 **Same Thing Using Exception Handling *(Part 2 of 2)***

Sample Dialogue 1

```
Enter number of male dancers:
4
Enter number of female dancers:
6
Each man must dance with 1.5 women.
Begin the lesson.
```

Sample Dialogue 2

```
Enter number of male dancers:
0
Enter number of female dancers:
0
Lesson is canceled. No students.
```

Note that this dialog and the dialogs below do not say `"Begin the lesson"`.

Sample Dialogue 3

```
Enter number of male dancers:
0
Enter number of female dancers:
5
Lesson is canceled. No men.
```

Sample Dialogue 4

```
Enter number of male dancers:
4
Enter number of female dancers:
0
Lesson is canceled. No women.
```

The created Exception object is not assigned to a variable, but rather is used as an (anonymous) argument to the throw operator. (Anonymous arguments were discussed in Chapter 4.) The keyword throw is an operator with syntax similar to the unary + or unary – operators. To make it look more like an operator, you can write it with parentheses around the argument, as follows:

```
throw (new Exception("Lesson is canceled. No men."));
```

Although it is perfectly legal and sensible to include these extra parentheses, nobody does include them.

Exception class

To understand this process of throwing, you need to know two things: What is this Exception class? And what does the throw operator do with the Exception object? The class Exception is another class from the standard Java package java.lang. As you have already seen, the class Exception has a constructor that takes a single String argument. The Exception object created stores this String argument (in a private instance variable). As you will see, this String argument can later be retrieved from the Exception object.

throw operator

catch block

getMessage

The throw operator causes a change in the flow of control and it delivers the Exception object to a suitable place, as we are about to explain. When the throw operator is executed, the try block ends immediately and control passes to the following catch block. (If it helps, you can draw an analogy between the execution of the throw operator in a try block and the execution of a break statement in a loop or switch statement.) When control is transferred to the catch block, the Exception object that is thrown is plugged in for the catch block parameter e. So, the expression e.getMessage() returns the string "Lesson is canceled. No men.". The method getMessage() of the class Exception is an accessor method that retrieves the String in the private instance variable of the Exception object—that is, the String used as an argument to the Exception constructor.

To see if you get the basic idea of how this exception throwing mechanism works, study the sample dialogs. The next few sections explain this mechanism in more detail.

try-throw-catch MECHANISM

try block

The basic way of handling exceptions in Java consists of the try-throw-catch trio. The general setup consists of a try block followed by one or more catch blocks. First let's describe what a try block is. A try **block** has the syntax

```
try
{
    Some_Code
}
```

This try block contains the code for the basic algorithm that tells what to do when everything goes smoothly. It is called a try block because it "tries" to execute the case where all goes smoothly.

Now, if something exceptional does happen, you want to throw an exception, which is a way of indicating that something unusual happened. So the basic outline, when we add a throw, is as follows:

```
try
{
    Code_That_May_Throw_An_Exception
}
```

The following is an example of a try block with throw statements included (copied from Display 9.2):

```
try
{
    if (men == 0 && women == 0)
        throw new Exception("Lesson is canceled. No students.");
    else if (men == 0)
        throw new Exception("Lesson is canceled. No men.");
    else if (women == 0)
        throw new Exception("Lesson is canceled. No women.");

    // women >= 0 && men >= 0
    if (women >= men)
        System.out.println("Each man must dance with " +
                                women/(double)men + " women.");
    else
        System.out.println("Each woman must dance with " +
                                men/(double)women + " men.");
}
```

throw statement

This try block contains the following three throw statements:

```
throw new Exception("Lesson is canceled. No students.");
throw new Exception("Lesson is canceled. No men.");
throw new Exception("Lesson is canceled. No women.");
```

throwing an exception

The value thrown is an argument to the throw operator and is always an object of some exception class. The execution of a throw statement is called **throwing an exception**.

catch block

handling an exception

As the name suggests, when something is "thrown," something goes from one place to another place. In Java, what goes from one place to another is the flow of control as well as the exception object that is thrown. When an exception is thrown, the code in the surrounding try block stops executing and (normally) another portion of code, known as a catch **block**, begins execution. The catch block has a parameter, and the exception object thrown is plugged in for this catch block parameter. This executing of the catch block is called **catching the exception** or **handling the exception**. When an exception is thrown, it should ultimately be handled by (caught by) some catch block.

throw Statement

Syntax:

```
throw new Exception_Class_Name(Possibly_Some_Arguments);
```

When the throw statement is executed, the execution of the surrounding try block is stopped and (normally) control is transferred to a catch block. The code in the catch block is executed next. See the box entitled "try-throw-catch" later in this chapter for more details.

Example:

```
throw new Exception("Division by zero.");
```

In Display 9.2, the appropriate catch block immediately follows the try block. We repeat the catch block in what follows:

```
catch(Exception e)
{
    String message = e.getMessage();
    System.out.println(message);
    System.exit(0);
}
```

This catch block looks very much like a method definition that has a parameter of a type Exception. It is not a method definition, but in some ways, a catch block is like a method. It is a separate piece of code that is executed when your program encounters (and executes) the following (within the preceding try block):

```
throw new Exception(Some_String);
```

So, this throw statement is similar to a method call, but instead of calling a method, it calls the catch block and says to execute the code in the catch block. A catch block is often referred to as an **exception handler**.

exception handler

Let's focus on the identifier e in the following line from a catch block:

```
catch(Exception e)
```

catch block parameter

That identifier e in the catch block heading is called the catch **block parameter**. Each catch block can have at most one catch block parameter. The catch block parameter does two things:

1. The catch block parameter is preceded by an exception class name that specifies what type of thrown exception object the catch block can catch.

2. The catch block parameter gives you a name for the thrown object that is caught, so you can write code in the catch block that does things with the thrown object that is caught.

Although the identifier e is often used for the catch block parameter, this is not required. You may use any non-keyword identifier for the catch block parameter just as you can for a method parameter.

The getMessage Method

Every exception has a String instance variable that contains some message, which typically identifies the reason for the exception. For example, if the exception is thrown as follows:

```
throw new Exception(String_Argument);
```

then the string given as an argument to the constructor Exception is used as the value of this String instance variable. If the object is called e, then the method call e.getMessage() returns this string.

Example:

Suppose the following throw statement is executed in a try block:

```
throw new Exception("Input must be positive.");
```

And suppose the following is a catch block immediately following the try block:

```
catch(Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Program aborted.");
    System.exit(0);
}
```

In this case, the method call e.getMessage() returns the string
"Input must be positive."

Let's consider two possible cases of what can happen when a try block is executed: (1.) no exception is thrown in the try block, and (2.) an exception is thrown in the try block and caught in the catch block. (Later we will describe a third case where the catch block does not catch the exception.)

If no exception is thrown, the code in the try block is executed to the end of the try block, the catch block is skipped, and execution continues with the code placed after the catch block.

If an exception is thrown in the try block, the rest of the code in the try block is skipped and (in simple cases) control is transferred to a following catch block. The thrown object is plugged in for the catch block parameter and the code in the catch block is executed. And then (provided the catch block code does not end the program or do something else to end the catch block code prematurely), the code that follows that catch block is executed.

catch Block Parameter

The catch block parameter is an identifier in the heading of a catch block that serves as a placeholder for an exception that might be thrown. When a (suitable) exception is thrown in the preceding try block, that exception is plugged in for the catch block parameter. The identifier e is often used for catch block parameters, but this is not required. You can use any legal (non-keyword) identifier for a catch block parameter.

Syntax:

```
  catch(Exception_Class_Name  Catch_Block_Parameter)
  {
      <Code to be performed if an exception of the named exception class is thrown in the try
 block.>
  }
```

You may use any legal identifier for the *Catch_Block_Parameter.*

Example:

In the following, e is the catch block parameter.

```
 catch(Exception e)
 {
     System.out.println(e.getMessage());
     System.out.println("Aborting program.");
     System.exit(0);
}
```

try-throw-catch

When used together, the try, throw, and catch statements are the basic mechanism for throwing and catching exceptions. The throw statement throws the exception. The catch block catches the exception. The throw statement is normally included in a try block. When the exception is thrown, the try block ends and then the code in the catch block is executed. After the catch block is completed, the code after the catch block(s) is executed (provided the catch block has not ended the program or performed some other special action).

If no exception is thrown in the try block, then after the try block is completed, program execution continues with the code after the catch block(s). (In other words, if no exception is thrown, the catch block(s) are ignored.)

SYNTAX:

```
try
{
    Some_Statements
   <Either some code with a throw statement or
              a method invocation that might throw an exception.>
    Some_More_Statements
}
catch(Exception_Class_Name  Catch_Block_Parameter)
{
    <Code to be performed if an exception of the named exception
                                class is thrown in the try block.>
}
```

You may use any legal identifier for the *Catch_Block_Parameter*; a common choice is e. The code in the `catch` block may refer to the *Catch_Block_Parameter*. If there is an explicit `throw` statement, it is usually embedded in an `if` statement or an `if-else` statement. There may be any number of `throw` statements and/or any number of method invocations that may throw exceptions. Each `catch` block can list only one exception, but there can be more than one `catch` block.

EXAMPLE:

See Display 9.2.

EXCEPTION CLASSES

There are more exception classes than just the single class `Exception`. There are more exception classes in the standard Java libraries and you can define your own exception classes. All the exception classes in the Java libraries have—and the exception classes you define should have—the following properties:

There is a constructor that takes a single argument of type `String`.

The class has an accessor method `getMessage()` that can recover the string given as an argument to the constructor when the exception object was created.

EXCEPTION CLASSES FROM STANDARD PACKAGES

Numerous predefined exception classes are included in the standard packages that come with Java. The names of predefined exceptions are designed to be self-explanatory. Some sample predefined exceptions are

```
IOException
NoSuchMethodException
FileNotFoundException
```

IOException

You have already had some experience with the IOException class, which is in the java.io package and requires that you import the IOException class with

```
import java.io.IOException;
```

The kind of use we have been making of the IOException class will be explained in Section 9.2 of this chapter.

Exception

The predefined exception class Exception is the root class for all exceptions. Every exception class is a descendant of the class Exception (that is, it is derived directly from the class Exception or from a class that is derived from the class Exception, or it arises from some longer chain of derivations ultimately starting with the class Exception). You can use the class Exception itself, just as we did in Display 9.2, but you are even more likely to use it to define a derived class of the class Exception. The class Exception is in the java.lang package and so requires no import statement.

The Class Exception

Every exception class is a descendent class of the class Exception. You can use the class Exception itself in a class or program, but you are even more likely to use it to define a derived class of the class Exception. The class Exception is in the java.lang package and so requires no import statement.

Self-Test Exercises

1. What output is produced by the following code?

```
int waitTime = 46;

  try
  {
      System.out.println("Try block entered.");
      if (waitTime > 30)
          throw new Exception("Over 30.");
      else if (waitTime < 30)
          throw new Exception("Under 30.");
      else
          System.out.println("No exception.");
      System.out.println("Leaving try block.");
  }
  catch(Exception thrownObject)
  {
      System.out.println(thrownObject.getMessage());
  }
  System.out.println("After catch block");
```

2. Suppose that in exercise 1 the line

   ```
   int waitTime = 46;
   ```

   were changed to

   ```
   int waitTime = 12;
   ```

   How would this affect the output?

3. In the code given in exercise 1, what are the throw statements?

4. What happens when a throw statement is executed? This is a general question. Tell what happens in general, not simply what happens in the code in exercise 1 or some other sample code.

5. In the code given in exercise 1, what is the try block?

6. In the code given in exercise 1, what is the catch block?

7. In the code given in exercise 1, what is the catch block parameter?

8. Is the following legal?

   ```
   IOException exceptionObject =
                       new IOException("Nothing to read!");
   ```

9. Is the following legal?

   ```
   IOException exceptionObject =
                       new IOException("Nothing to read!");
   throw exceptionObject;
   ```

DEFINING EXCEPTION CLASSES

A throw statement can throw an exception object of any exception class. A common thing to do is to define an exception class whose objects can carry the precise kinds of information you want thrown to the catch block. An even more important reason for defining a specialized exception class is so that you can have a different type to identify each possible kind of exceptional situation.

Every exception class you define must be a derived class of some already defined exception class. An exception class can be a derived class of any exception class in the standard Java libraries or of any exception class that you have already successfully defined. Our examples will be derived classes of the class Exception.

constructors

When defining an exception class, the constructors are the most important members. Often there are no other members, other than those inherited from the base class. For example, in Display 9.3, we've defined an exception class called DivisionByZeroException whose only members are a no-argument constructor and a constructor with one String parameter. In most cases, these two constructors are all the exception

Display 9.3 A Programmer-Defined Exception Class

```
public class DivisionByZeroException extends Exception
{
    public DivisionByZeroException()
    {
        super("Division by Zero!");
    }

    public DivisionByZeroException(String message)
    {
        super(message);
    }
}
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

class definition contains. However, the class does inherit all the methods of the class Exception.[1] In particular, the class DivisionByZeroException inherits the method getMessage, which returns a string message. In the no-argument constructor, this string message is set with the following, which is the first line in the no-argument constructor definition:

```
super("Division by Zero!");
```

This is a call to a constructor of the base class Exception. As we have already noted, when you pass a string to the constructor for the class Exception, it sets the value of a String instance variable that can later be recovered with a call to getMessage. The method getMessage is an ordinary accessor method of the class Exception. The class DivisionByZeroException inherits this String instance variable as well as the accessor method getMessage.

For example, in Display 9.4, we give a sample program that uses this exception class. The exception is thrown using the no-argument constructor, as follows:

```
throw new DivisionByZeroException();
```

This exception is caught in the catch block shown in Display 9.4. Consider the following line from that catch block:

```
System.out.println(e.getMessage());
```

[1] Some programmers would prefer to derive the DivisionByZeroException class from the predefined class ArithmeticException, but that would make it a kind of exception that you are not required to catch in your code, so you would lose the help of the compiler in keeping track of uncaught exceptions. For more details, see the subsection “Exceptions to the Catch or Declare Rule” later in this chapter. If this footnote does not make sense to you, you can safely ignore it.

Display 9.4 **Using a Programmer-Defined Exception Class *(Part 1 of 3)***

We will present an improved version of this program later in this chapter.

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DivisionDemoFirstVersion
{

    public static void main(String[] args) throws IOException
    {
        try
        {
            BufferedReader keyboard = new BufferedReader(
                             new InputStreamReader(System.in));

            System.out.println("Enter numerator:");
            String numeratorString = keyboard.readLine();
            int numerator = Integer.parseInt(numeratorString);
            System.out.println("Enter denominator:");
            String denominatorString = keyboard.readLine();
            int denominator =
                             Integer.parseInt(denominatorString);

            if (denominator == 0)
                throw new DivisionByZeroException();

            double quotient = numerator/(double)denominator;
            System.out.println(numerator + "/"
                                     + denominator
                                     + " = " + quotient);
        }
        catch(DivisionByZeroException e)
        {
            System.out.println(e.getMessage());
            secondChance();
        }

        System.out.println("End of program.");
    }

    public static void secondChance() throws IOException
    {
        BufferedReader keyboard = new BufferedReader(
                             new InputStreamReader(System.in));
```

Display 9.4 **Using a Programmer-Defined Exception Class *(Part 2 of 3)***

```
        System.out.println("Try again:");
        System.out.println("Enter numerator:");
        String numeratorString = keyboard.readLine();
        int numerator = Integer.parseInt(numeratorString);
        System.out.println("Enter denominator:");
        System.out.println("Be sure the denominator is not zero.");
        String denominatorString = keyboard.readLine();
        int denominator = Integer.parseInt(denominatorString);

        if (denominator == 0)
        {
            System.out.println("I cannot do division by zero.");
            System.out.println("Aborting program.");
            System.exit(0);
        }

        double quotient = ((double)numerator)/denominator;
        System.out.println(numerator + "/"
                                     + denominator
                                     + " = " + quotient);
    }

}
```

Sometimes it is better to handle an exceptional case without throwing an exception.

Sample Dialogue 1

```
Enter numerator:
11
Enter denominator:
5
11/5 = 2.2
End of program.
```

This line produces the following output to the screen in Sample Dialogs 2 and 3:

```
Division by Zero!
```

The definition of the class `DivisionByZeroException` in Display 9.3 has a second constructor with one parameter of type `String`. This constructor allows you to choose any message you like when you throw an exception. If the `throw` statement in Display 9.4 had instead used the string argument

```
throw new DivisionByZeroException(
                        "Oops. Shouldn't divide by zero.");
```

Display 9.4 **Using a Programmer-Defined Exception Class** ***(Part 3 of 3)***

Sample Dialogue 2

```
Enter numerator:
11
Enter denominator:
0
Division by Zero!
Try again.
Enter numerator:
11
Enter denominator:
Be sure the denominator is not zero.
5
11/5 = 2.2
End of program.
```

Sample Dialogue 3

```
Enter numerator:
11
Enter denominator:
0
Division by Zero!
Try again.
Enter numerator:
11
Enter denominator:
Be sure the denominator is not zero.
0
I cannot do division by zero.
Aborting program.
```

then in Sample Dialogs 2 and 3, the statement

```
System.out.println(e.getMessage());
```

would have produced the following output to the screen:

```
Oops. Shouldn't divide by zero.
```

Notice that in Display 9.4, the `try` block is the normal part of the program. If all goes normally, that is the only code that will be executed, and the dialog will be like the one shown in Sample Dialog 1. In the exceptional case, when the user enters a zero for a denominator, the exception is thrown and then is caught in the catch block. The catch block outputs the message of the exception and then calls the method secondChance. The method secondChance gives the user a second chance to enter the input correctly and then carries out the calculation. If the user tries a second time to divide by zero, the method ends the program. The method secondChance is there only for this exceptional case. So, we have separated the code for the exceptional case of a division by zero into a separate method, where it will not clutter the code for the normal case.

Tip

An Exception Class Can Carry a Message of Any Type

It is possible to define your exception classes so they have constructors that take arguments of other types that are stored in instance variables. In such cases you would define accessor methods for the value stored in the instance variable. For example, if that is desired, you can have an exception class that carries an int as a message. In that case, you would need a new accessor method name, perhaps getBadNumber(). An example of one such exception class is given in Display 9.5.

Tip

Preserve getMessage

For all predefined exception classes, getMessage will return the string that is passed as an argument to the constructor (or will return a default string if no argument is used with the constructor). For example, if the exception is thrown as follows:

```
throw new Exception("Wow, this is exceptional!");
```

then "Wow, this is exceptional!" is used as the value of the String instance variable of the object created. If the object is called e, the method invocation e.getMessage() returns "Wow, this is exceptional!" You want to preserve this behavior in the exception classes you define.

For example, suppose you are defining an exception class named NegativeNumberException. Be sure to include a constructor with a string parameter that begins with a call to super, as illustrated by the following constructor:

```
public NegativeNumberException(String message)
{
    super(message);
}
```

Display 9.5 **An Exception Class with an `int` Message**

```java
public class BadNumberException extends Exception
{
    private int badNumber;

    public BadNumberException(int number)
    {
        super("BadNumberException");
        badNumber = number;
    }

    public BadNumberException()
    {
        super("BadNumberException");
    }

    public BadNumberException(String message)
    {
        super(message);
    }

    public int getBadNumber()
    {
        return badNumber;
    }
}
```

The call to super is a call to a constructor of the base class. If the base class constructor handles the message correctly, then so will a class defined in this way.

You should also include a no-argument constructor in each exception class. This no-argument constructor should set a default value to be retrieved by getMessage. The constructor should begin with a call to super, as illustrated by the following constructor:

```java
public NegativeNumberException()
{
    super("Negative Number Exception!");
}
```

If getMessage works as we described for the base class, then this sort of no-argument constructor will work correctly for the new exception class being defined. A full definition of the class NegativeNumberException is given in Display 9.7.

Exception Object Characteristics

The two most important things about an exception object are its type (the exception class) and a message that it carries in an instance variable of type `String`. This string can be recovered with the accessor method `getMessage`. This string allows your code to send a message along with an exception object, so that the `catch` block can use the message.

Self-Test Exercises

10. Define an exception class called `PowerFailureException`. The class should have a constructor with no parameters. If an exception is thrown with this zero-argument constructor, `getMessage` should return `"Power Failure!"` The class should also have a constructor with a single parameter of type `String`. If an exception is thrown with this constructor, then `getMessage` returns the value that was used as an argument to the constructor.

11. Define an exception class called `TooMuchStuffException`. The class should have a constructor with no parameters. If an exception is thrown with this zero-argument constructor, `getMessage` should return `"Too much stuff!"`. The class should also have a constructor with a single parameter of type `String`. If an exception is thrown with this constructor, then `getMessage` returns the value that was used as an argument to the constructor.

12. Suppose the exception class `ExerciseException` is defined as follows:

```
public class ExerciseException extends Exception
{
    public ExerciseException()
    {
        super("Exercise Exception thrown!");
        System.out.println("Exception thrown.");
    }

    public ExerciseException(String message)
    {
        super(message);
        System.out.println(
          "ExerciseException invoked with an argument.");
    }
}
```

What output would be produced by the following code (which is just an exercise and not likely to occur in a program)?

```
ExerciseException e =
               new ExerciseException("Do Be Do");
System.out.println(e.getMessage());
```

extra code on CD

The class `ExerciseException` is on the CD that comes with this text.

Programmer-Defined Exception Classes

You may define your own exception classes, but every such class must be a derived class of an already existing exception class (either from one of the standard Java libraries or programmer defined).

Guidelines:

- If you have no compelling reason to use any other class as the base class, use the class Exception as the base class.
- You should define two (or more) constructors, as described later in this list.
- Your exception class inherits the method getMessage. Normally, you do not need to add any other methods, but it is legal to do so.
- You should start each constructor definition with a call to the constructor of the base class, such as the following:

  ```
  super("Sample Exception thrown!");
  ```

- You should include a no-argument constructor, in which case the call to super should have a string argument that indicates what kind of exception it is. This string can then be recovered by using the getMessage method.
- You should also include a constructor that takes a single string argument. In this case, the string should be an argument in a call to super. That way, the string can be recovered with a call to getMessage.

Example:

```
public class SampleException extends Exception
{
    public SampleException()
    {
        super("Sample Exception thrown!");
    }

    public SampleException(String message)
    {
        super(message);
    }
}
```

The class SampleException is on the CD that comes with this text.

extra code on CD

13. Suppose the exception class `TestException` is defined as follows:

```
public class TestException extends Exception
{
    public TestException()
    {
        super("Test Exception thrown!");
        System.out.println(
                      "Test exception thrown!!");
    }

    public TestException(String message)
    {
        super(message);
        System.out.println(
         "Test exception thrown with an argument!");
    }

    public void testMethod()
    {
        System.out.println("Message is " + getMessage());
    }
}
```

What output would be produced by the following code (which is just an exercise and not likely to occur in a program)?

```
TestException exceptionObject = new TestException();
System.out.println(exceptionObject.getMessage());
exceptionObject.testMethod();
```

extra code on CD

The class `TestException` is on the CD that comes with this text.

14. Suppose the exception class `MyException` is defined as follows:

```
public class MyException extends Exception
{
    public MyException()
    {
        super("My Exception thrown!");
    }

    public MyException(String message)
    {
        super("MyException: " + message);
    }
}
```

What output would be produced by the following code (which is just an exercise and not likely to occur in a program)?

```
int number;
try
{
    System.out.println("try block entered:");
    number = 42;
    if (number > 0)
        throw new MyException("Hi Mom!");
    System.out.println("Leaving try block.");
}

catch(MyException exceptionObject)
{
    System.out.println(exceptionObject.getMessage());
}
System.out.println("End of example.");
```

The class MyException is on the CD that comes with this text.

extra code on CD

15. Suppose that in exercise 14 the catch block were changed to the following. (The type MyException is replaced with Exception.) How would this affect the output?

```
catch(Exception exceptionObject)
{
    System.out.println(exceptionObject.getMessage());
}
```

16. Suppose that in exercise 14 the line

```
    number = 42;
```

were changed to

```
    number = -58;
```

How would this affect the output?

17. Although an exception class normally carries only a string message, you can define exception classes to carry a message of any type. For example, objects of the following type can also carry a double "message" (as well as a string message):

```
public class DoubleException extends Exception
{
    private double doubleMessage;
    public DoubleException()
    {
        super("DoubleException thrown!");
    }

    public DoubleException(String message)
    {
        super(message);
    }

    public DoubleException(double number)
    {
        super("DoubleException thrown!");
        doubleMessage = number;
    }

    public double getNumber()
    {
        return doubleMessage;
    }
}
```

What output would be produced by the following code (which is just an exercise and not likely to occur in a program)?

```
DoubleException e =
               new DoubleException(41.9);

System.out.println(e.getNumber());

System.out.println(e.getMessage());
```

extra code on CD

The class `DoubleException` is on the CD that comes with this text.

18. Can you define an exception class as a derived class of the predefined class `IOException`, or must a defined exception class be derived from the class `Exception`?

MULTIPLE catch BLOCKS

A `try` block can potentially throw any number of exception values, and they can be of differing types. In any one execution of the `try` block, at most one exception will be thrown (since a `throw` statement ends the execution of the `try` block), but different types of exception values can be thrown on different occasions when the `try` block is executed. Each `catch` block can only catch values of the exception class type given in the `catch` block heading. However, you can catch exception values of differing types by placing more than one `catch` block after a `try` block. For example, the program in Display 9.6 has two `catch` blocks after its `try` block. The class `NegativeNumberException`, which is used in that program, is given in Display 9.7.

Display 9.6 Catching Multiple Exceptions *(Part 1 of 2)*

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class MoreCatchBlocksDemo
{
   public static void main(String[] args) throws IOException
   {
       BufferedReader keyboard =
                  new BufferedReader(new InputStreamReader(System.in));

       try
       {
           System.out.println("How many pencils do you have?");
           String pencilString = keyboard.readLine();
           int pencils = Integer.parseInt(pencilString);

           if (pencils < 0)
                throw new NegativeNumberException("pencils");

           System.out.println("How many erasers do you have?");
           String eraserString = keyboard.readLine();
           int erasers = Integer.parseInt(eraserString);
           double pencilsPerEraser;

           if (erasers < 0)
               throw new NegativeNumberException("erasers");
           else if (erasers != 0)
               pencilsPerEraser = pencils/(double)erasers;
           else
               throw new DivisionByZeroException();

           System.out.println("Each eraser must last through "
                + pencilsPerEraser + " pencils.");
       }

       catch(NegativeNumberException e)
       {
           System.out.println("Cannot have a negative number of "
               + e.getMessage());
       }
       catch(DivisionByZeroException e)
       {
          System.out.println("Do not make any mistakes.");
       }
```

Display 9.6 Catching Multiple Exceptions *(Part 2 of 2)*

```
            System.out.println("End of program.");
        }
    }
```

Sample Dialogue 1

```
How many pencils do you have?
5
How many erasers do you have?
2
Each eraser must last through 2.5 pencils
End of program.
```

Sample Dialogue 2

```
How many pencils do you have?
−2
Cannot have a negative number of pencils
End of program.
```

Sample Dialogue 3

```
How many pencils do you have?
5
How many erasers do you have?
0
Do not make any mistakes.
End of program.
```

Pitfall

Catch the More Specific Exception First

When catching multiple exceptions, the order of the `catch` blocks can be important. When an exception is thrown in a `try` block, the `catch` blocks are examined in order, and the first one that matches the type of the exception thrown is the one that is executed. Thus, the following ordering of catch blocks would not be good:

```
catch (Exception e)
{
```

```
          .
          .
          .
}
catch(NegativeNumberException e)
{
          .
          .
          .
}
```

The second catch block can never be reached.

With this ordering, the catch block for NegativeNumberException would never be used, because all exceptions are caught by the first catch block. Fortunately, the compiler will warn you about this. The correct ordering is to reverse the catch blocks so that the more specific exception comes before its parent exception class, as shown in the following:

```
catch(NegativeNumberException e)
{
          .
          .
          .
}
catch(Exception e)
{
          .
          .
          .
}
```

Display 9.7 The Class NegativeNumberException

```
public class NegativeNumberException extends Exception
{
    public NegativeNumberException()
    {
        super("Negative Number Exception!");
    }

    public NegativeNumberException(String message)
    {
        super(message);
    }
}
```

Self-Test Exercises

19. What output will be produced by the following code? (The definition of the class `NegativeNumberException` is given in Display 9.7.)

```
int n;
try
{
    n = 42;
    if (n > 0)
        throw new Exception();
    else if (n < 0)
        throw new NegativeNumberException();
    else
        System.out.println("Bingo!");
}

catch(NegativeNumberException e)
{
    System.out.println("First catch.");
}
catch(Exception e)
{
    System.out.println("Second catch.");
}
System.out.println("End of exercise.");
```

20. Suppose that in exercise 19 the line

```
n = 42;
```

were changed to

```
n = −42;
```

How would this affect the output?

21. Suppose that in exercise 19 the line

```
n = 42;
```

were changed to

```
n = 0;
```

How would this affect the output?

9.2 Throwing Exceptions in Methods

buck *n. Games. A counter or marker formerly passed from one poker player to another to indicate an obligation, especially one's turn to deal.*

The American Heritage Dictionary of the English Language, Third Edition

The buck stops here.

Harry S Truman (sign on Truman's desk while he was president)

So far our examples of exception handling have been toy examples. We have not yet shown any examples of a program that makes good and realistic use of exception handling. However, now you know enough about exception handling to discuss more realistic uses of exception handling. This section explains the single most important exception handling technique, namely throwing an exception in a method and catching it outside the method.

THROWING AN EXCEPTION IN A METHOD

Sometimes it makes sense to throw an exception in a method but not catch it in the method. For example, you might have a method with code that throws an exception if there is an attempt to divide by zero, but you may not want to catch the exception in that method. Perhaps some programs that use that method should simply end if the exception is thrown, and other programs that use the method should do something else. So, you would not know what to do with the exception if you caught it inside the method. In such cases, it makes sense to not catch the exception in the method definition, but instead to have any program (or other code) that uses the method place the method invocation in a `try` block and catch the exception in a `catch` block that follows that `try` block.

Look at the program in Display 9.8. It has a `try` block, but there is no `throw` statement visible in the `try` block. The statement that does the throwing in that program is

```
if (bottom == 0)
   throw new DivisionByZeroException();
```

This statement is not visible in the `try` block. However, it is in the `try` block in terms of program execution, because it is in the definition of the method `safeDivide`, and there is an invocation of `safeDivide` in the `try` block.

The meaning of `throws DivisionByZero` in the heading of `safeDivide` is discussed in the next subsection.

Display 9.8 **Use of a throws Clause *(Part 1 of 2)***

We will present an even better version of this program later in this chapter.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DivisionDemoSecondVersion
{
   public static void main(String[] args) throws IOException
   {
       BufferedReader keyboard =
                  new BufferedReader(new InputStreamReader(System.in));

        try
        {
           System.out.println("Enter numerator:");
           String numeratorString = keyboard.readLine();
           int numerator = Integer.parseInt(numeratorString);
           System.out.println("Enter denominator:");
           String denominatorString = keyboard.readLine();
           int denominator =
                        Integer.parseInt(denominatorString);

           double quotient = safeDivide(numerator, denominator);
           System.out.println(numerator + "/"
                                     + denominator
                                     + " = " + quotient);
        }
        catch(DivisionByZeroException e)
        {
            System.out.println(e.getMessage());
            secondChance();
        }

        System.out.println("End of program.");
   }


  public static double safeDivide(int top, int bottom)
                             throws DivisionByZeroException
   {
       if (bottom == 0)
           throw new DivisionByZeroException();

       return top/(double)bottom;
   }
```

Display 9.8 **Use of a throws Clause *(Part 2 of 2)***

```
    public static void secondChance() throws IOException
    {
        BufferedReader keyboard =
                  new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Try again.");
        System.out.println("Enter numerator:");
        String numeratorString = keyboard.readLine();
        int numerator = Integer.parseInt(numeratorString);
        System.out.println("Enter denominator:");
        System.out.println("Be sure the denominator is not zero.");
        String denominatorString = keyboard.readLine();
        int denominator = Integer.parseInt(denominatorString);

        if (denominator == 0)
        {
            System.out.println("I cannot do division by zero.");
            System.out.println("Aborting program.");
            System.exit(0);
        }

        double quotient = numerator/(double)denominator;
        System.out.println(numerator + "/"
                                      + denominator
                                      + " = " + quotient);
    }
}
```

The input/output dialogs are identical to those for the program in Display 9.4.

DECLARING EXCEPTIONS IN A throws CLAUSE

If a method does not catch an exception, then (in most cases) it must at least warn programmers that any invocation of the method might possibly throw an exception. This warning is called a *throws clause,* and including an exception class in a throws clause is called **declaring the exception**. For example, a method that might possibly throw a DivisionByZeroException and that does not catch the exception would have a heading similar to the following:

throws clause

declaring an exception

```
public void sampleMethod() throws DivisionByZeroException
```

The part throws DivisionByZeroException is a throws **clause** stating that an invocation of the method sampleMethod might throw a DivisionByZeroException.

throws clause

If there is more than one possible exception that can be thrown in the method definition, then the exception types are separated by commas, as illustrated in what follows:

```
public void sampleMethod()
             throws DivisionByZeroException, SomeOtherException
```

Most "ordinary" exceptions that might be thrown when a method is invoked must be accounted for in one of two ways:

1. The possible exception can be caught in a `catch` block within the method definition.
2. The possible exception can be declared at the start of the method definition by placing the exception class name in a `throws` clause (and letting whoever uses the method worry about how to handle the exception).

Catch or Declare Rule

This is often called the **Catch or Declare Rule**. In any one method, you can mix the two alternatives, catching some exceptions and declaring others in a `throws` clause.

You already know about technique 1, handling exceptions in a `catch` block. Technique 2 is a form of shifting responsibility ("passing the buck"). For example, suppose `yourMethod` has a `throws` clause as follows:

```
public void yourMethod() throws DivisionByZeroException
```

In this case, `yourMethod` is absolved of the responsibility of catching any exceptions of type `DivisionByZeroException` that might occur when `yourMethod` is executed. If, however, there is another method, `myMethod`, that includes an invocation of `yourMethod`, then `myMethod` must handle the exception. When you add a `throws` clause to `yourMethod`, you are saying to `myMethod`, "If you invoke `yourMethod`, you must handle any `DivisionByZeroException` that is thrown." In effect, `yourMethod` has passed the responsibility for any exceptions of type `DivisionByZeroException` from itself to any method that calls it.

Of course, if `yourMethod` passes responsibility to `myMethod` by including `DivisionByZeroException` in a `throws` clause, then `myMethod` may also pass the responsibility to whoever calls it by including the same `throws` clause in its definition. But in a well-written program, every exception that is thrown should eventually be caught by a `catch` block in some method that does not just declare the exception class in a `throws` clause.

When an exception is thrown in a method but not caught in that method, that immediately ends the method invocation.

Be sure to note that the `throws` clause for a method is for exceptions that "get outside" the method. If they do not get outside the method, they do not belong in the `throws` clause. If they get outside the method, they belong in the `throws` clause no matter where they originate. If an exception is thrown in a `try` block that is inside a method definition and is caught in a `catch` block inside the method definition, then its exception class need not be listed in the `throws` clause. If a method definition includes an invocation of another method and that other method can throw an exception that is not caught, then the exception class of that exception should be placed in the `throws` clause.

throws CLAUSE

If you define a method that might throw exceptions of some particular class, then normally either your method definition must include a `catch` block that will catch the exception or you must declare (that is, list) the exception class within a `throws` clause, as described in what follows.

SYNTAX (COVERS MOST COMMON CASES):

```
public Type_Or_void Method(Parameter_List) throws List_Of_Exceptions
Body_Of_Method
```

EXAMPLE:

```
public void yourMethod(int n) throws IOException, MyException
{
        .
        .
        .
}
```

THROWING AN EXCEPTION CAN END A METHOD

If a method throws an exception, and the exception is not caught inside the method, then the method invocation ends immediately after the exception is thrown.

In Display 9.8, we have rewritten the program from Display 9.4 so that the exception is thrown in the method `safeDivide`. The method `main` includes a call to the method `safeDivide` and puts the call in a `try` block. Because the method `safeDivide` can throw a `DivisionByZeroException` that is not caught in the method `safeDivide`, we needed to declare this in a `throws` clause at the start of the definition of `safeDivide`. If we set up our program in this way, the case in which nothing goes wrong is completely isolated and easy to read. It is not even cluttered by `try` blocks and `catch` blocks.

throws CLAUSE IN DERIVED CLASSES

When you override a method definition in a derived class, it should have the same exception classes listed in its `throws` clause that it had in the base class, or it should have a `throws` clause whose exceptions are a subset of those in the base class `throws` clause. Put another way, when you override a method definition, you cannot add any exceptions to the `throws` clause (but you can delete some exceptions if you want). This makes sense, since an object of the derived class can be used anyplace an object of the base class can be used, and so an overridden method must fit any code written for an object of the base class.

What Happens If an Exception Is Never Caught?

If every method up to and including the main method simply includes a throws clause for a particular class of exceptions, then it may turn out that an exception of that class is thrown but never caught. In such cases, when an exception is thrown but never caught, either the program ends or its performance may become unreliable. For GUI programs, such as those that use JOptionPane or the other Swing classes we discuss in Chapter 16, if an exception is thrown but never caught, then nothing happens, but if your code does not somehow account for the thrown exception, then the user may be left in an unexplained situation. If your program does not use GUI classes and an exception is thrown but never caught, then the program ends with an error message giving the name of the exception class.

In a well-written program, every exception that is thrown should eventually be caught by a catch block in some method.

WHEN TO USE EXCEPTIONS

So far, most of our examples of exception handling have been unrealistically simple. A better guideline for how you should use exceptions is to separate throwing an exception and catching the exception into separate methods. In most cases, you should include any throw statement within a method definition, list the exception class in a throws clause for that method, and place the catch block in *a different method*. In outline form, the technique is as follows:

```
public void yourMethod() throws YourException
{
        ...
    throw new YourException(<Maybe an argument.>);
        ...
}
```

Then, when yourMethod is used by some otherMethod, the otherMethod must account for the exception. For example:

```
public void otherMethod()
{
        ...
    try
    {
            ...
        yourMethod();
            ...
    }
    catch(YourException e)
```

```
        {
           <Handle exception.>
        }
            ...
  }
```

Even this kind of use of a throw statement should be reserved for cases where it is unavoidable. *If you can easily handle a problem in some other way, do not throw an exception. Reserve* throw *statements for situations in which the way the exceptional condition is handled depends on how and where the method is used.* If the way that the exceptional condition is handled depends on how and where the method is invoked, then the best thing to do is to let the programmer who invokes the method handle the exception. In all other situations, it is preferable to avoid throwing exceptions. Let's outline a sample scenario of this kind of situation.

Suppose you are writing a library of methods to deal with patient monitoring systems for hospitals. One method might compute the patient's average daily temperature by accessing the patient's record in some file and dividing the sum of the temperatures by the number of times the temperature was taken. Now suppose these methods are used for creating different systems to be used in different situations. What should happen if the patient's temperature was never taken and so the averaging would involve a divides by zero? In an intensive-care unit, this would indicate something is very wrong. So for that system, when this potential division by zero would occur, an emergency message should be sent out. However, for a system that is to be used in a less urgent setting, such as outpatient care or even in some noncritical wards, it might have no significance and so a simple note in the patient's record would suffice. In this scenario, the method for doing the averaging of the temperatures should throw an exception when this division by zero occurs, list the exception in the throws clause, and let each system handle the exception case in the way that is appropriate to that system.

When to Throw an Exception

Exceptions should be reserved for situations where a method has an exceptional case and different invocations of the method would handle the exceptional case differently. In this situation, you would throw an exception in the method definition, not catch the exception in the method, but list the exception in the throws clause for the method.

IOException

You should now be able to figure out why we use a throws clause with IOException in any methods that use BufferedReader to read keyboard input, such as in the following from Display 9.8:

```
public static void main(String[] args) throws IOException
```

or the following method heading from the same program:

```
public static void secondChance() throws IOException
```

Most of the methods that deal with input using `BufferedReader` can throw an `IOException`. So, when you do console input using `BufferedReader`, the method heading must include the class `IOException` in a `throws` clause, or else the method must have `catch` blocks that catch any possible `IOException`. This will be discussed in more detail in Chapter 10. However, we do note here that a well-written program would catch any such `IOException` in a suitable `catch` block. In Display 9.9 we have rewritten the program in Display 9.8 so that all `IOException`s are caught.

EVENT-DRIVEN PROGRAMMING

event-driven programming

firing an event

Exception handling is our first example of a programming methodology known as **event-driven programming**. With event-driven programming, objects are defined so that they send **events**, which are themselves objects, to other objects that handle the events. Sending the event is called **firing the event**. In exception handling, the event objects are the exception objects. They are fired (thrown) by an object when the object invokes a method that throws the exception. An exception event is sent to a `catch` block, where it is handled. Of course, a `catch` block is not exactly an object, but the idea is the same. Also, our programs have mixed event-driven programming (exception handling) with more traditional programming techniques. When we study how you construct windowing systems using the Swing libraries (Chapter 16), you will see examples of programming where the dominant technique is event-driven programming.

Self-Test Exercises

22. What is the output produced by the following program?

```
public class Exercise
{
    public static void main(String[] args)
    {
       try
       {
           System.out.println("Trying");
           sampleMethod(98.6);
           System.out.println("Trying after call.");
       }
       catch(Exception e)
```

Display 9.9 **Catching an IOException *(Part 1 of 2)***

Yes, we will give another, better version of this program later in the chapter.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DivisionDemoThirdVersion
{
   public static void main(String[] args)
   {

        BufferedReader keyboard =
                  new BufferedReader(new InputStreamReader(System.in));

        try
        {
           System.out.println("Enter numerator:");
           String numeratorString = keyboard.readLine();
           int numerator = Integer.parseInt(numeratorString);
           System.out.println("Enter denominator:");
           String denominatorString = keyboard.readLine();
           int denominator =
                        Integer.parseInt(denominatorString);


           double quotient = safeDivide(numerator, denominator);
           System.out.println(numerator + "/"
                                        + denominator
                                        + " = " + quotient);
        }
        catch(DivisionByZeroException e)
        {
            System.out.println(e.getMessage());
            secondChance();
        }
        catch(IOException e)
        {
            System.out.println("IO Problem. Aborting program.");
            System.exit(0);
        }
        System.out.println("End of program.");
    }
```

No throws clause

This constructor invocation does not throw any exceptions.

The readLine method may throw an IOException.

safeDivide may throw a DivisionByZeroException

Display 9.9 Catching an IOException *(Part 2 of 2)*

```
   public static double safeDivide(int top, int bottom)
                             throws DivisionByZeroException
   {
       if (bottom == 0)
           throw new DivisionByZeroException();

       return top/(double)bottom;
   }

   public static void secondChance()
   {
      BufferedReader keyboard =
                    new BufferedReader(new InputStreamReader(System.in));

      try
      {
         System.out.println("Try again.");
         System.out.println("Enter numerator:");
         String numeratorString = keyboard.readLine();
         int numerator = Integer.parseInt(numeratorString);
         System.out.println("Enter denominator:");
         System.out.println(
                        "Be sure the denominator is not zero.");
         String denominatorString = keyboard.readLine();
         int denominator = Integer.parseInt(denominatorString);

         if (denominator == 0)
         {
             System.out.println("I cannot do division by zero.");
             System.out.println("Aborting program.");
             System.exit(0);
         }

         double quotient = numerator/(double)denominator;
         System.out.println(numerator + "/" + denominator
                                        + " = " + quotient);
      }
      catch(IOException e)
      {
          System.out.println("IO Problem. Aborting program.");
          System.exit(0);
      }
   }
}
```

The input/output dialogs are identical to those for the program in Display 9.4.

No throws clause

The readLine method may throw an IOException.

```
       {
           System.out.println("Catching.");
       }

       System.out.println("End program.");
    }

    public static void sampleMethod(double test)
                                              throws Exception
    {
           System.out.println("Starting sampleMethod.");
           if (test < 100)
                   throw new Exception();
    }
}
```

The class `Exercise` is on the CD that comes with this text.

extra code on CD

23. Suppose that in exercise 22 the line

```
    sampleMethod(98.6);
```

in the `try` block were changed to

```
    sampleMethod(212);
```

How would this affect the output?

24. Correct the following method definition by adding a suitable `throws` clause:

```
public static void doStuff(int n)
{
    if (n < 0)
        throw new Exception("Negative number.");
}
```

25. What happens if an exception is thrown inside a method invocation but the exception is not caught inside the method?

26. Suppose there is an invocation of method A inside of method B, and an invocation of method B inside of method C. When method C is invoked, this leads to an invocation of method B, and that in turn leads to an invocation of method A. Now, suppose that method A throws an exception but does not catch it within A. Where might the exception be caught? In B? In C? Outside of C?

9.3 More Programming Techniques for Exception Handling

Only use this in exceptional circumstances.

Warren Peace, *The Lieutenant's Tool*

In this section we present a number of the finer points about programming with exception handling in Java. We also define and explain a class called `ConsoleIn` for doing simple keyboard input.

Pitfall

Nested try-catch Blocks

You can place a `try` block and its following `catch` blocks inside a larger `try` block or inside a larger `catch` block. On rare occasions this may be useful, but it is almost always better to place the inner `try catch` blocks inside a method definition and place an invocation of the method in the outer `try` or `catch` block (or maybe just eliminate one or more `try` blocks completely).

If you place a `try` block and its following `catch` blocks inside a larger `catch` block, you will need to use different names for the `catch` block parameters in the inner and outer blocks. This has to do with how Java handles nested blocks of any kind. Remember, `try` blocks and `catch` blocks are blocks.

If you place a `try` block and its following `catch` blocks inside a larger `try` block, and an exception is thrown in the inner `try` block but is not caught in the inner `catch` blocks, then the exception is thrown to the outer `try` block for processing and might be caught in one of its `catch` blocks.

THE finally BLOCK ✤

The `finally` block contains code to be executed whether or not an exception is thrown in a `try` block. The `finally` block, if used, is placed after a `try` block and its following `catch` blocks. The general syntax is as follows:

```
try
{
   ...
}
catch(ExceptionClass1 e)
{
   ...
}
.
.
.
catch(ExceptionClassLast e)
{
   ...
}
finally
{
    < Code to be executed whether or not an exception is thrown or caught.>
}
```

Now, suppose that the `try-catch-finally` blocks are inside a method definition. (After all, every set of `try-catch-finally` blocks is inside of some method, even if it is only the method `main`.) There are three possibilities when the code in the `try-catch-finally` blocks is run:

1. The `try` block runs to the end and no exception is thrown. In this case, the `finally` block is executed after the `try` block.
2. An exception is thrown in the `try` block and is caught in one of the `catch` blocks positioned after the `try` block. In this case, the `finally` block is executed after the `catch` block is executed.
3. An exception is thrown in the `try` block and there is no matching `catch` block in the method to catch the exception. In this case, the method invocation ends and the exception object is thrown to the enclosing method. However, the `finally` block is executed before the method ends. Note that you cannot account for this last case simply by placing code after the `catch` blocks.

RETHROWING AN EXCEPTION ✜

A `catch` block can contain code that throws an exception. In rare cases you may find it useful to catch an exception and then, depending on the string produced by `getMessage` or depending on something else, decide to throw the same or a different exception for handling further up the chain of exception handling blocks.

Self-Test Exercises

27. Can you have a `try` block and corresponding `catch` blocks inside another larger `try` block?

28. Can you have a `try` block and corresponding `catch` blocks inside another larger `catch` block?

29. ✜ What is the output produced by the following program? What would the output be if the argument to `exerciseMethod` were –42 instead of 42? What would it be if the argument

were 0 instead of 42? (The class NegativeNumberException is defined in Display 9.7, but you need not review that definition to do this exercise.)

```
public class FinallyDemo
{
    public static void main(String[] args)
    {
        try
        {
            exerciseMethod(42);
        }
        catch(Exception e)
        {
            System.out.println("Caught in main.");
        }
    }

    public static void exerciseMethod(int n) throws Exception
    {
        try
        {
            if (n > 0)
                throw new Exception();
            else if (n < 0)
                throw new NegativeNumberException();
            else
               System.out.println("No Exception.");
            System.out.println("Still in sampleMethod.");
        }
        catch(NegativeNumberException e)
        {
            System.out.println("Caught in sampleMethod.");
        }

        finally
        {
            System.out.println("In finally block.");
        }
        System.out.println("After finally block.");
    }
}
```

extra code on CD

The class FinallyDemo is on the CD that comes with this text.

EXCEPTIONS TO THE CATCH OR DECLARE RULE

As we already noted, in most "ordinary" cases, an exception must either be caught in a `catch` block or declared in a `throws` clause. That is the Catch or Declare Rule, but there are exceptions to this rule. There are some classes whose exceptions you do not need to account for in this way (although you can catch them in a `catch` block if you want to). These are typically exceptions that result from errors of some sort. They usually indicate that your code should be fixed, not that you need to add a `catch` block. You do not write a `throw` statement for these exceptions. They are often thrown by methods in standard library classes, but it would be legal to throw one of these exceptions in the code you write.

Exceptions that are descendents of the class `RuntimeException` do not need to be accounted for in a `catch` block or `throws` clause. There is also another category of classes that are called `Error` classes and that behave like exception classes in that they can be thrown and caught in a `catch` block. However, you are not required to account for `Error` objects in a `catch` block or `throws` clause. The situation is diagrammed as a class hierarchy in Display 9.10. All the classes shown in blue follow the Catch or Declare Rule, which says that if their objects are thrown, then they must either be caught in a `catch` block or declared in a `throws` clause. All the classes shown in yellow are exempt from the Catch or Declare Rule.

checked and unchecked exceptions

Exception classes that follow the Catch or Declare Rule are often called **checked exceptions**. Exceptions that are exempt from the Catch or Declare Rule are often called **unchecked exceptions**.

You need not worry too much about which exceptions you do and do not need to declare in a `throws` clause. If you fail to account for some exception that Java requires you to account for, the compiler will tell you about it, and you can then either catch it or declare it in a `throws` clause.

Catch or Declare Rule

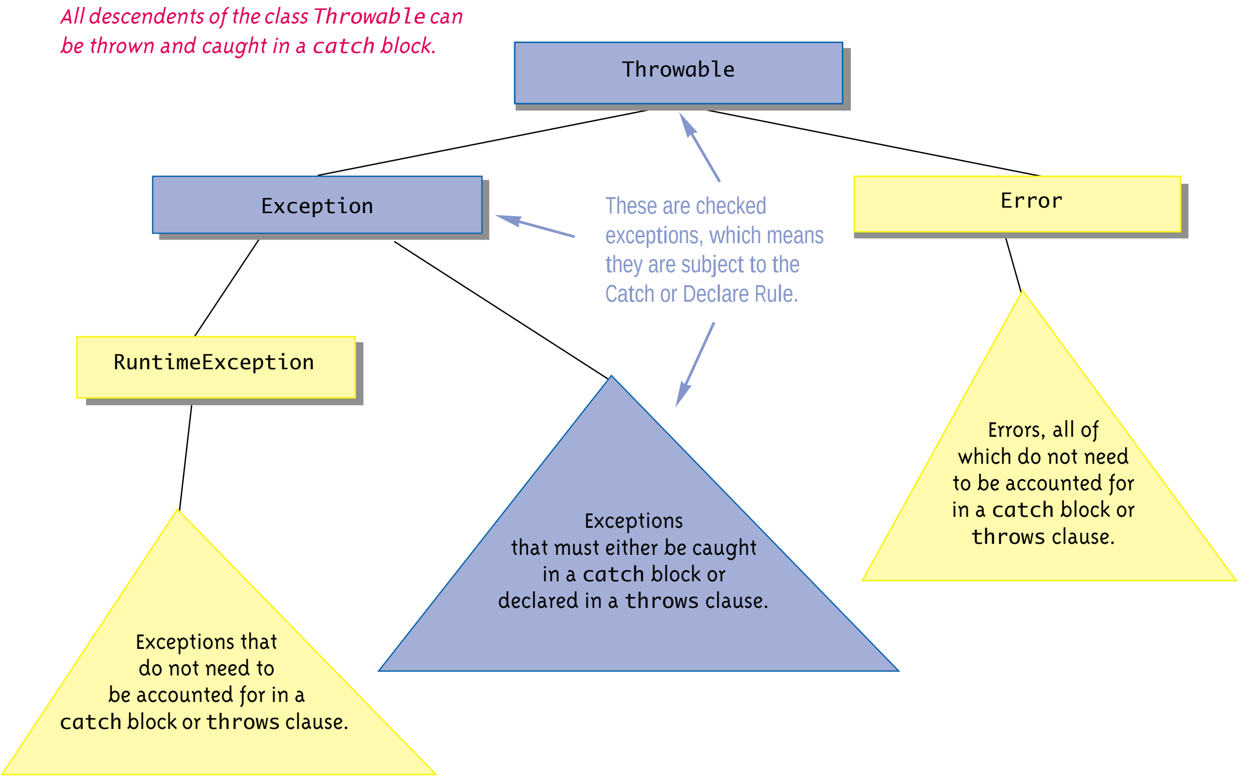
Most "ordinary" exceptions that might be thrown when a method is invoked must be accounted for in one of two ways:

1. The possible exception can be caught in a `catch` block within the method definition.
2. The possible exception can be declared at the start of the method definition by placing the exception class name in a `throws` clause.

This rule is called the **Catch or Declare Rule**.

Exceptions that must follow the Catch or Declare Rule are often called **checked exceptions**. Display 9.10 explains which methods are checked exceptions (and so must follow the Catch or Declare Rule) and which throwable objects are exempt from the Catch or Declare Rule.

Display 9.10 **Hierarchy of Throwable Objects**


All descendents of the class Throwable can be thrown and caught in a catch block.
Throwable
Exception
Error
These are checked exceptions, which means they are subject to the Catch or Declare Rule.
RuntimeException
Exceptions that must either be caught in a catch block or declared in a throws clause.
Errors, all of which do not need to be accounted for in a catch block or throws clause.
Exceptions that do not need to be accounted for in a catch block or throws clause.

THE AssertionError CLASS ✜

When we discussed the `assert` operator and assertion checking in Chapter 3, we said that if your program contains an assertion check and the assertion check fails, your program will end with an error message. That statement is more or less true, but it is incomplete. What happens is that an object of the class `AssertionError` is thrown. If it is not caught in a `catch` block, your program ends with an error message. However, if you wish, you can catch it in a `catch` block, although that is not a very common thing to do. The `AssertionError` class is in the `java.lang` package and so requires no import statement.

As the name suggests, the class `AssertionError` is derived from the class `Error`, so you are not required to either catch it in a `catch` block or declare it in a `throws` clause.

NumberFormatException

Methods that process String arguments as if they were numbers will often throw a NumberFormatException if a String argument does not correctly represent a Java number of the requisite type. NumberFormatException is in the standard Java package java.lang and so requires no import statement. NumberFormatException is a descendent class of RuntimeException, so you need not account for a NumberFormatException by catching it in a catch block or declaring it in a throws clause. However, you are allowed to catch a NumberFormatException in a catch block and that can sometimes be useful. For example, the method Integer.parseInt will throw a NumberFormatException if its argument is not a well-formed Java int numeral string. The method inputInt in Display 9.11 prompts the user for an int input value and uses Integer.parseInt to convert the input string to an int value. If the input string is not a correctly formed int numeral string, then a NumberFormatException is thrown and caught in a catch block, which causes the surrounding loop body to be repeated so that the user is asked to reenter the input.

Tip

Exception Controlled Loops

Sometimes when an exception is thrown, such as a NumberFormatException for an ill-formed input string, you want your code to simply repeat some code so that the user (or whatever) can get things right on a second or subsequent try. One way to set up your code to repeat a loop every time a particular exception is thrown is as follows:

```
boolean done = false;

while (! done)
{
    try
    {
        <Code that may throw an exception in the class Exception_Class.>
        done = true; //Will end the loop.
        <Possibly more code.>
    }
    catch(Exception_Class e)
    {
         <Some code.>
    }
}
```

Note that if an exception is thrown in the first piece of code in the try block, then the try block ends before the line that sets done to true is executed and so the loop body is repeated. If no exception is thrown, then done is set to true and the loop body is not repeated.

Display 9.11 Catching a NumberFormatException *(Part 1 of 3)*

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DivisionDemoFinalVersion
{
    public static void main(String[] args)
    {
        int numerator = inputInt("Enter numerator:");
        int denominator = 0; //to keep compiler happy
        boolean done = false;

        while (! done)
        {
            try
            {
                 denominator = inputInt("Enter denominator:");
                 double quotient = safeDivide(numerator, denominator);
                 done = true;
                 System.out.println(numerator + "/" + denominator
                                       + " = " + quotient);
             }
             catch(DivisionByZeroException e)
             {
                 System.out.println("Cannot divide by zero.");
                 System.out.println("Try again.");
             }
        }

        System.out.println("End of program.");
    }

    public static double safeDivide(int top, int bottom)
                             throws DivisionByZeroException
    {
        if (bottom == 0)
            throw new DivisionByZeroException();
        return top/(double)bottom;
    }
    public static int inputInt(String prompt)
    {
        BufferedReader keyboard =
                  new BufferedReader(new InputStreamReader(System.in));
```

safeDivide can throw a DivisionByZeroException.

Not executed if safeDivide throws an exception.

Display 9.11 **Catching a NumberFormatException *(Part 2 of 3)***

```
        String numeralString = null; //to keep compiler happy
        int inputValue = 0; //to keep compiler happy
        boolean done = false;

        while (! done)
        {
            try
            {
                System.out.println(prompt);
                numeralString = keyboard.readLine();
                inputValue = Integer.parseInt(numeralString);
                done = true;
            }
            catch(NumberFormatException e)
            {
                System.out.println(numeralString
                                            + " is not a valid input.");
                System.out.println("Try again.");
            }
            catch(IOException e)
            {
                System.out.println("IO Problem. Aborting program.");
                System.exit(0);
            }
        }

      return inputValue;

   }

}
```

readLine can throw an IOException.

Not executed if an exception is thrown.

parseInt can throw a NumberFormatException.

SAMPLE DIALOGUE 1

```
Enter numerator:
2000
Enter denominator:
1000
2000/1000 = 2.0
End of program.
```

Display 9.11 **Catching a NumberFormatException *(Part 3 of 3)***

Sample Dialogue 2

```
Enter numerator:
2,000
2,000 is not a valid input.
Try again.
Enter numerator:
2,000.00
2,000.00 is not a valid input.
Try again.
Enter numerator:
2 thousand
2 thousand is not a valid input.
Try again.
Enter numerator:
2000
Enter denominator:
0
Cannot divide by zero.
Try again.
Enter denominator:
1,000
1,000 is not a valid input.
Try again.
Enter denominator:
1000
2000/1000 = 2.0
End of program.
```

Display 9.11 contains two examples of such loops, one for DivisionByZeroException and one for NumberFormatException. Minor variations on this outline can accommodate a range of different situations for which you want to repeat code on throwing an exception.

ArrayIndexOutOfBoundsException

You should read Section 6.1 of Chapter 6, which covers array basics, before reading this short subsection. If you have not yet covered some of Chapter 6, you can omit this section and return to it at a later time.

If your program attempts to use an array index that is out of bounds, an ArrayIndexOutOfBoundsException is thrown and your program ends, unless the exception is caught in a catch block. ArrayIndexOutOfBoundsException is a descendent of the class

RuntimeException and so need not be caught or accounted for in a throws clause. This sort of exception normally indicates that there is something wrong with your code and means that you need to fix your code, not catch an exception. Thus, an ArrayIndexOutOfBoundsException normally functions more like a run-time error message than a regular exception.

ArrayIndexOutOfBoundsException is in the standard Java package java.lang and so requires no import statement should you decide to use it by name.

Self-Test Exercises

30. Would you get a compiler error message if you omit the following catch block from the method inputInt in Display 9.11? If you would not get a compiler error message, would you sometimes get a run-time error message? Explain your answers.

```
catch(NumberFormatException e)
{
    System.out.println(numeralString
                      + " is not a valid input.");
    System.out.println("Try again.");
}
```

31. Would you get a compiler error message if you omit the following catch block from the method inputInt in Display 9.11? If you would not get a compiler error message, would you sometimes get a run-time error message? Explain your answers.

```
catch(IOException e)
{
    System.out.println("IO Problem. Aborting program.");
    System.exit(0);
}
```

Example

Console Input Class

Java has no class to gracefully handle simple console input (that is, text input entered at the keyboard). However, it is easy to define such a class. One such class is the class ConsoleIn given in Display 9.12. Now that you understand exception handling, you can easily understand the definition of ConsoleIn. Use of the class ConsoleIn was described in an stared section of Chapter 2. In this Programming Example subsection, we explain details of the code for ConsoleIn.

constructor

The class ConsoleIn uses the object System.in to do its input and then processes the input so that it passes the data on to your program as values of type int, double, or other standard

Display 9.12 **Definition of the Class ConsoleIn *(Part 1 of 4)***

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

/**
 Class for robust keyboard input.
 If the user enters an improper input, that is, an input of the wrong type
 or a blank line when the line should be nonblank, then the user is given
 an error message and is prompted to reenter the input.
*/
public class ConsoleIn
{
    private static final BufferedReader inputObject =
             new BufferedReader(new InputStreamReader(System.in));

    /**
     Reads a line of text and returns that line as a String value.
     This will read the rest of a line if the line is already
     partially read.
    */
    public static String readLine()
    {
        String inputLine = null;

        try
        {
            inputLine = inputObject.readLine();
        }
        catch(IOException e)
        {
            System.out.println("Fatal Error.Aborting.");
            System.exit(0);
        }

        return inputLine;
    }
```

Display 9.12 Definition of the Class ConsoleIn *(Part 2 of 4)*

```
    /**
     The user is supposed to enter a whole number of type int on a line by
     itself. There may be whitespace before and/or after the number.
     Returns the number entered as a value of type int. The rest of the line
     is discarded. If the input is not entered correctly, then in most cases,
     the user will be asked to reenter the input. In particular, incorrect
     number formats and blank lines result in a prompt to reenter the input.
    */
    public static int readLineInt()
    {
        String inputString = null;
        int number = 0;//To keep the compiler happy.
        boolean done = false;

        while (! done)
        {
            try
            {
                inputString = readLine();
                number = Integer.parseInt(inputString.trim());
                done = true;
            }
            catch (NumberFormatException e)
            {
                System.out.println(
                         "Input number is not in correct format.");
                System.out.println("The input number must be");
                System.out.println("a whole number written as an");
                System.out.println("ordinary numeral, such as 42.");
                System.out.println("Do not include a plus sign.");
                System.out.println("Minus signs are OK,");

                System.out.println("Try again.");
                System.out.println("Enter a whole number:");
            }
        }

        return number;
    }
```

Display 9.12 Definition of the Class ConsoleIn *(Part 3 of 4)*

```java
/**
 The user is supposed to enter a number of type double on a line by itself.
 There may be whitespace before and/or after the number.
 Returns the number entered as a value of type double. The rest of the line
 is discarded. If the input is not entered correctly, then in most cases,
 the user will be asked to reenter the input. In particular, incorrect
 number formats and blank lines result in a prompt to reenter the input.
*/
public static double readLineDouble()
{
    String inputString = null;
    double number = 0;//To keep the compiler happy.
    boolean done = false;

    while (! done)
    {
        try
        {
            inputString = readLine();
            number = Double.parseDouble(inputString.trim());
            done = true;
        }
        catch (NumberFormatException e)
        {
            System.out.println("Input number is not in correct format.");
            System.out.println("The input number must be");
            System.out.println("an ordinary number either with");
            System.out.println("or without a decimal point,");
            System.out.println("such as 42 or 41.999");
            System.out.println("Try again.");
            System.out.println("Enter a number:");
        }
    }

    return number;
}
```

<The methods readLineFloat, readLineByte, readLineShort, and readLineLong are minor variations on readLineInt and readLineDouble with only the obvious changes for each number type. The complete code for all these methods is given in Appendix 5.>

Display 9.12 **Definition of the Class ConsoleIn *(Part 4 of 4)***

```
    /**
     Returns the first nonwhitespace character on the input line. The rest of the line
     is discarded. If the line contains only whitespace, the user is asked to reenter.
    */
    public static char readLineNonwhiteChar()
    {
        boolean done = false;
        String inputString = null;
        char nonWhiteChar = ' ';//To keep the compiler happy.

        while (! done)
        {
            inputString = readLine();
            inputString = inputString.trim();
            if (inputString.length() == 0)
            {
                System.out.println("Input is not correct.");
                System.out.println("The input line must contain at");
                System.out.println("least one non-whitespace character.");
                System.out.println("Try again.");
                System.out.println("Enter input:");
            }
            else
            {
                nonWhiteChar = inputString.charAt(0);
                done = true;
            }
        }

        return nonWhiteChar;
    }

    /**
     Input should consist of a single word on a line, possibly surrounded by
     whitespace. The line is read and discarded. If the input word is "true" or
    "t", then true is returned. If the input word is "false" or "f", then false
     is returned. Uppercase and lowercase letters are considered equal. If the
     user enters anything else, the user is asked to reenter the input.
    */
    public static boolean readLineBoolean()
     <The rest of the definition of the method readLineBoolean is Self-Test Exercise 32.>
}
```

types. In previous programs (most recently in Display 9.11), we've used input objects of the following form:

```
BufferedReader keyboard =
                   new BufferedReader(new InputStreamReader(System.in));
```

The class ConsoleIn uses just such an input object, but the input object is a private static variable, declared and initialized as follows:

```
private static final BufferedReader inputObject =
              new BufferedReader(new InputStreamReader(System.in));
```

All input is then read by static methods that use this inputObject.

readLine

The method readLine of the class ConsoleIn is invoked as follows:

```
someStringVariable = ConsoleIn.readLine();
```

Inside the definition of the method readLine, the actual reading of an input line is done by the object inputObject as follows:

```
inputLine = inputObject.readLine();
```

What is new in the readLine method of ConsoleIn is that the programmer using the readLine in ConsoleIn need not worry about IOExceptions. All possible IOExceptions are caught inside the method readLine of ConsoleIn.

readLineInt

The method readLineInt obtains the int value input as follows:

```
inputString = readLine();
number = Integer.parseInt(inputString.trim());
```

Then the int value number is returned. This is the same way that we read int values in previous programs, most recently in Display 9.11. In fact, the method readLineInt of the class ConsoleIn is almost the same as the method inputInt in Display 9.11. Like the method inputInt, the method readLineInt catches any NumberFormatException and asks the user to reenter the input if an exception is thrown.

One thing that we added to readLineInt that was not in the method inputInt (Display 9.11) is the invocation of the String method trim(). This invocation of trim() removes extra leading and trailing whitespace and so makes a NumberFormatException less likely.

readLineDouble

The method readLineDouble is essentially the same as readLineInt except that it uses Double.parseDouble instead of Integer.parseInt. The methods readLineFloat, readLineByte, readLineShort, and readLineLong are also similar minor variations on readLineInt. In fact, they are so similar that we have not given the code for them in Display 9.12. You should easily be able to write the code using readLineInt as a model, but if need be you can find the missing code in Appendix 5.

readLineBoolean

The method readLineBoolean is left as a routine Self-Test Exercise (Self-Test Exercise 32).

The method readLineNonwhiteChar reads the entire line and removes the leading and trailing whitespace as follows:

```
inputString = readLine();
inputString = inputString.trim();
```

readLine–NonwhiteChar

The method readLineNonwhiteChar needs to return only the first character in inputString. It uses the String method charAt to obtain that character as follows:

```
nonWhiteChar = inputString.charAt(0);
```

The value nonWhiteChar is then returned by readLineNonwhiteChar.

This discussion of readLineNonwhiteChar has so far assumed that the user entered at least one nonwhitespace character. If the user enters only whitespace characters, then the string inputString will have length zero. The method readLineNonwhiteChar tests for length zero and asks the user to reenter the input in that case.

A sample use of the class ConsoleIn is given in Display 9.13. An addition example can be found in Display 2.9 of Chapter 2.

Display 9.13 **Keyboard Input with ConsoleIn *(Part 1 of 2)***

```
public class ConsoleInDemo
{
   public static void main(String[] args)
   {
     char ans;

     do
     {
          System.out.println("Enter width of lot in feet:");
          double width = ConsoleIn.readLineDouble();

          System.out.println("Enter length of lot in feet:");
          double length = ConsoleIn.readLineDouble();

          double area = width*length;

          System.out.println("A lot that is " + width
                                              + " feet wide");
          System.out.println("and " + length + " feet long");
          System.out.println("has an area of " + area
                                        + " square feet.");
```

The file ConsoleIn.class must be in the same directory as this program.

Display 9.13 **Keyboard Input with ConsoleIn *(Part 2 of 2)***

```
          System.out.println("Repeat calculation? (y/n)");
          ans = ConsoleIn.readLineNonwhiteChar();
     } while (Character.toLowerCase(ans) == 'y');

     System.out.println("End of program.");
   }
}
```

Sample Dialogue

```
Enter width of lot in feet:
50.7 ft.
Input number is not in correct format
The input number must be
an ordinary number either with
or without a decimal point,
such as 42 or 42.999
Try again.
Enter a number:
50.7
Enter length of lot in feet:
100.6
A lot that is 50.7 feet wide
and 100.6 feet long
has an area of 5100.42 square feet.
Repeat Calculation? (y/n)
yes
Enter width of lot in feet:
50
Enter length of lot in feet:
100
A lot that is 50.0 feet wide
and 100.0 feet long
has an area of 5000.0 square feet.
Repeat Calculation? (y/n)
N
End of program.
```

Self-Test Exercises

32. Complete the definition of the method `readLineBoolean` of the class `ConsoleIn` in Display 9.12.

Chapter Summary

- Exception handling allows you to design and code the normal case for your program separately from the code that handles exceptional situations.
- An exception can be thrown in a `try` block. Alternatively, an exception can be thrown in a method definition that does not include a `try` block (or does not include a `catch` block to catch that type of exception). In this case, an invocation of the method can be placed in a `try` block.
- An exception is caught in a `catch` block.
- A `try` block must be followed by at least one `catch` block and can be followed by more than one `catch` block. If there are multiple `catch` blocks, always list the `catch` block for a more specific exception class before the `catch` block for a more general exception class.
- The best use of exceptions is to throw an exception in a method (but not catch it in the method), but to only do this when the way the exception is handled will vary from one invocation of the method to another. There is seldom any other situation that can profitably benefit from throwing an exception.
- If an exception is thrown in a method but not caught in that method, then, in most "ordinary" cases, the exception type must be listed in the `throws` clause for that method.

Answers to Self-Test Exercises

1.
```
Try block entered.
Over 30.
After catch block
```

2. The output would then be

```
Try block entered.
Under 30.
After catch block
```

3. There are two `throw` statements:

```
throw new Exception("Over 30.");
throw new Exception("Under 30.");
```

4. When a `throw` statement is executed, that is the end of the enclosing `try` block. No other statements in the `try` block are executed, and control passes to the following `catch` block(s). When we say that control passes to the following `catch` block, we mean that the exception object that is thrown is plugged in for the `catch` block parameter and the code in the `catch` block is executed.

5.

```
try
{
    System.out.println("Try block entered.");
    if (waitTime > 30)
        throw new Exception("Over 30.");
    else if (waitTime < 30)
        throw new Exception("Under 30.");
    else
        System.out.println("No exception.");
    System.out.println("Leaving try block.");
}
```

6.

```
catch(Exception thrownObject)
{
    System.out.println(thrownObject.getMessage());
}
```

7. `thrownObject`

8. Yes, it is legal.

9. Yes, it is legal.

10.

```
public class PowerFailureException extends Exception
{
    public PowerFailureException()
    {
        super("Power Failure!");
    }

    public PowerFailureException(String message)
    {
        super(message);
    }
}
```

11.

```
public class TooMuchStuffException extends Exception
{
    public TooMuchStuffException()
    {
        super("Too much stuff!");
    }

    public TooMuchStuffException(String message)
    {
        super(message);
    }
}
```

12.
```
ExerciseException invoked with an argument.
Do Be Do
```

13.
```
Test exception thrown!!
Test Exception thrown!
Message is Test Exception thrown!
```

14.
```
try block entered:
MyException: Hi Mom!
End of example.
```

15. The output would be the same.

16. The output would then be

```
try block entered:
Leaving try block.
End of example.
```

17.
```
41.9
DoubleException thrown!
```

18. Yes, you can define an exception class as a derived class of the class `IOException`.

19.
```
Second catch.
End of exercise.
```

20. The output would then be

```
First catch.
End of exercise.
```

21. The output would then be

```
Bingo!
End of exercise.
```

22.
```
Trying
Starting sampleMethod.
Catching.
End program.
```

23. The output would then be

```
Trying
Starting sampleMethod.
Trying after call.
End program.
```

24.
```
public static void doStuff(int n) throws Exception
{
    if (n < 0)
        throw new Exception("Negative number.");
}
```

25. If a method throws an exception and the exception is not caught inside the method, then the method invocation ends immediately after the exception is thrown. If the method invocation is inside a `try` block, then the exception is thrown to a matching `catch` block, if there is one. If there is no `catch` block matching the exception, then the method invocation ends as soon as that exception is thrown.

26. It might be caught in method B. If it is not caught in method B, it might be caught in method C. If it is not caught in method C, it might be caught outside of method C.

27. Yes, you can have a `try` block and corresponding `catch` blocks inside another larger `try` block.

28. Yes, you can have a `try` block and corresponding `catch` blocks inside another larger `catch` block.

29.
```
Caught in main.
In finally block.
```

If the argument to `sampleMethod` were `–42` instead of `42`, the output would be

```
Caught in sampleMethod.
In finally block.
After finally block.
```

30. You would not get a compiler error message because the class `NumberFormatException` is a descendent of the class `RuntimeException` and so need not be caught in a `catch` block or accounted for in a `throws` clause. If you do eliminate this `catch` block and a `NumberFormatException` is thrown, then your program would give a run-time error message.

31. If you omit the `IOException catch` block from the method `inputInt` in Display 9.11, you would get a compiler error message saying that a possible `IOException` must be accounted for by either a `catch` block or by being listed in a `throws` clause.

32. See Appendix 5 for the complete definition.

Programming Projects

1. Write a program that converts dates from numerical month/day/year format to normal “month day, year” format. (for example, 12/25/2000 corresponds to December 25, 2000). You will define three exception classes, one called `MonthException`, another called

DayException, and a third called YearException. If the user enters anything other than a legal month number (integers from 1 to 12), your program will throw and catch a MonthException and ask the user to reenter the month. Similarly, if the user enters anything other than a valid day number (integers from 1 to either 28, 29, 30, or 31, depending on the month and year), then your program will throw and catch a DayException and ask the user to reenter the day. If the user enters a year that is not in the range 1000 to 3000 (inclusive), then your program will throw and catch a YearException and ask the user to reenter the year. (There is nothing very special about the numbers 1000 and 3000 other than giving a good range of likely dates.) See Self-Test Exercise 20 in Chapter 4 for details on leap years.

2. Write a program that can serve as a simple calculator. This calculator keeps track of a single number (of type double) that is called result and that starts out as 0.0. Each cycle allows the user to repeatedly add, subtract, multiply, or divide by a second number. The result of one of these operations becomes the new value of result. The calculation ends when the user enters the letter R for "result" (either in uppercase or lowercase). The user is allowed to do another calculation from the beginning as often as he or she wants. Use the ConsoleIn class from Display 9.12 for input.

   The input format is shown in the following sample dialog. If the user enters any operator symbol other than +, –, *, or /, then an UnknownOperatorException is thrown and the user is asked to reenter that line of input. Defining the class UnknownOperatorException is part of this project.

```
Calculator is on.
result = 0.0
+5
result + 5.0 = 5.0
new result = 5.0
*2.2
result * 2.2 = 11.0
updated result = 11.0
% 10
% is an unknown operation.
Reenter, your last line:
* 0.1
result * 0.1 = 1.1
updated result = 1.1
r
Final result = 1.1
Again? (y/n)
yes
result = 0.0
+10
result + 10.0 = 10.0
new result = 10.0
/2
```

```
result / 2.0 = 5.0
updated result = 5.0
r
Final result = 5.0
Again? (y/n)
N
End of Program
```

For additional online Programming Projects, click the CodeMate icons below.

9.3 CodeMate